HADOOP WITH PYTHON TUTORIAL

Donald Miner
@donaldpminer
dminer@minerkasch.com

PyCon 2015
April 9th 2015

MINER & KASCH

Agenda

- Introduction to Hadoop
- MapReduce with mrjob
- Pig with Python UDFs
- snakebite for HDFS
- HBase and python clients
- Spark and PySpark

VM / Exercise Logistics

While I lecture for a bit:

- Download VirtualBox
- Start distributing VM (we'll be passing it around)
- Load up VM!

Ask your neighbor for help before you ask me

Hadoop

Distributed system for storage and compute

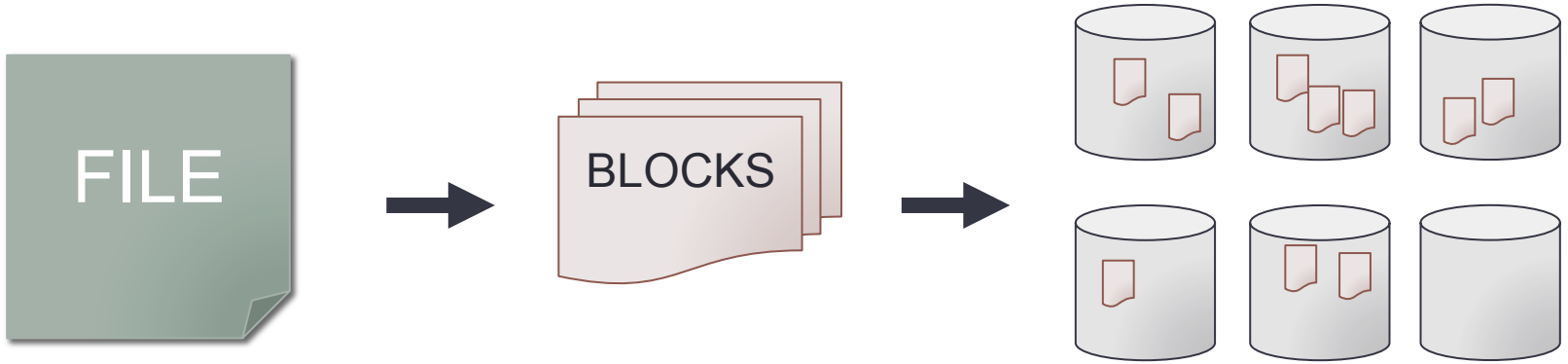
Hadoop

Distributed system for storage and compute

… built with Java

Hadoop Distributed File System (HDFS)

- Stores files in folders (that's it)
  - Nobody cares what's in your files
- Chunks large files into blocks (~64MB-2GB)
- 3 replicas of each block (better safe than sorry)
- Blocks are scattered all over the place



MapReduce

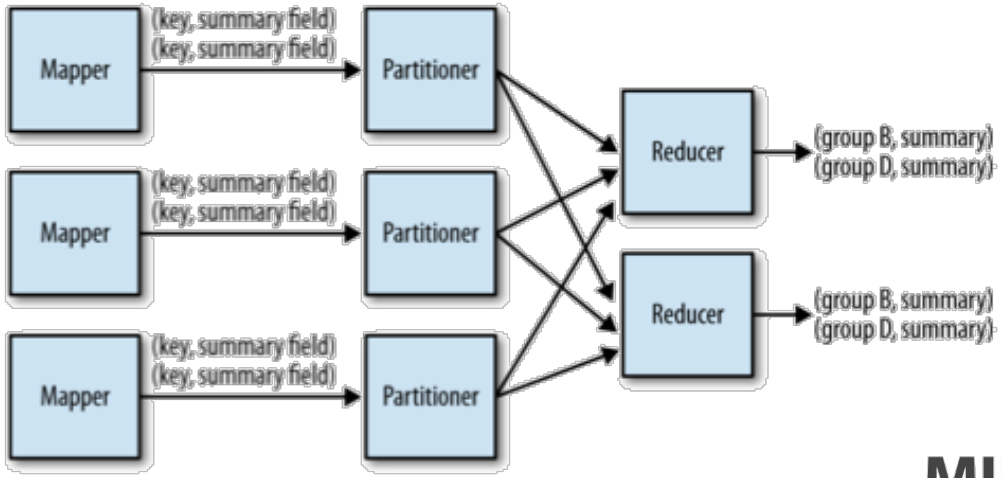
- Analyzes *raw* data in HDFS *where the data is*
- Jobs are split into Mappers and Reducers

Mappers (you code this)

Loads data from HDFS
Filter, transform, parse
Outputs (key, value) pairs

Reducers (you code this, too)

Automatically Groups by the mapper's output key
Aggregate, count, statistics
Outputs to HDFS



Hadoop Ecosystem

- Higher-level languages like Pig and Hive
- HDFS Data systems like HBase and Accumulo
- Alternative execution engines like Storm and Spark
- Close friends like ZooKeeper, Flume, Avro, Kafka

Cool Thing #1: Linear Scalability

- HDFS and MapReduce scale linearly
- If you have twice as many computers, jobs run twice as fast
- If you have twice as much data, jobs run twice as slow
- If you have twice as many computers, you can store twice as much data

DATA LOCALITY!!

Cool Thing #2: Schema on Read

BEFORE:

ETL, SCHEMA DESIGN UPFRONT, TOSSING OUT ORIGINAL DATA, COMPREHENSIVE DATA STUDY

WITH HADOOP:

LOAD DATA FIRST, ASK QUESTIONS LATER

Data is parsed/interpreted as it is loaded out of HDFS

What implications does this have?

Keep original data around!

Have multiple views of the same data!

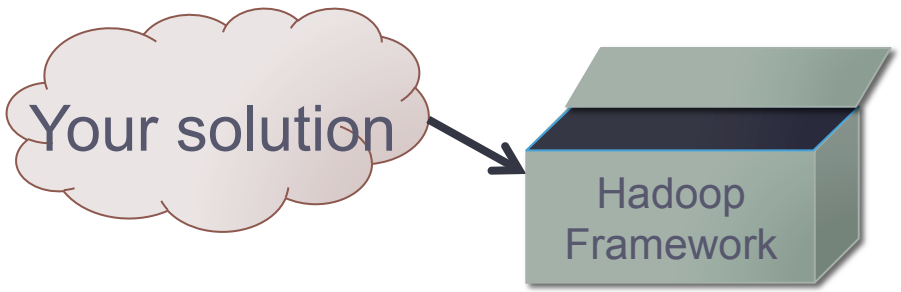
Work with unstructured data sooner!

Cool Thing #3: Transparent Parallelism

Code deployment?
Scalability?
Threading?
RPC?
Network programming?
Message passing?
Data storage?
Locking?
Distributed stuff?
Inter-process communication?
Fault tolerance?
Data center fires?

With MapReduce/HDFS, I DON'T CARE

… I just have to be sure my solution fits into this tiny box




```
class MRWordFreqCount(MRJob):

    def mapper(self, _, line):
        for word in WORD_RE.findall(line):
            yield word.lower(), 1

    def combiner(self, word, counts):
        yield word, sum(counts)

    def reducer(self, word, counts):
        yield word, sum(counts)
```

Cool Thing #4: Unstructured Data

- Unstructured data:
  media, text,
  forms, log data
  lumped structured data
- Query languages like SQL and Pig assume some sort of “structure”
- MapReduce is just Java (and Python):
  You can do anything Java can do in a Mapper or Reducer

Why Python?

- Python vs. Java
- Compiled vs. scripts
- Python libraries we all love
- Integration with other things

Why Not?

- Python vs. Java
- Almost nothing is native
  - Performance
  - Being out of date
  - Being “weird”
- Smaller community, almost no official support

Questions about Hadoop?

mrjob

- Write MapReduce jobs in Python!
- Open sourced and maintained by Yelp
- Wraps “Hadoop Streaming” in cpython Python 2.5+
- Well documented
- Can run locally, in Amazon EMR, or Hadoop

Canonical Word Count

```
from mrjob.job import MRJob
import re

WORD_RE = re.compile(r"[\w']+")

class MRWordFreqCount(MRJob):

    def mapper(self, _, line):
        for word in WORD_RE.findall(line):
            yield (word.lower(), 1)

    def reducer(self, word, counts):
        yield (word, sum(counts))


if __name__ == '__main__':
    MRWordFreqCount.run()
```

Canonical Word Count

```
from mrjob.job import MRJob
import re

WORD_RE = re.compile(r"[\w']+")
```

The quick brown fox jumps over the lazy dog

```
    def mapper(self, _, line):
        for word in WORD_RE.findall(line):
            yield (word.lower(), 1)

    def reducer(self, word, counts):
        yield (word, sum(counts))


if __name__ == '__main__':
    MRWordFreqCount.run()
```

the, 1
quick, 1
brown, 1
fox, 1
jumps, 1
over, 1
the, 1
lazy, 1
dog, 1

MINER & KASCH

Canonical Word Count

```
from mrjob.job import MRJob
import re

WORD_RE = re.compile(r"[\w']+")

class MRWordFreqCount(MRJob):

    def mapper(self, _, line):
        for word in WORD_RE.findall(line):
            yield (word.lower(), 1)

    def reducer(self, word, counts):
        yield (word, sum(counts))


if __name__ == '__main__':
    MRWordFreqCount.run()
```

I like this Hadoop thing

i, 1
like, 1
this, 1
hadoop, 1
thing, 1

Canonical Word Count

```
from mrjob.job import MRJob
import re

WORD_RE = re.compile(r"[\w']+")

class MRWordFreqCount(MRJob):

    def mapper(self, _, line):
        for word in WORD_RE.findall(line):
            yield (word.lower(), 1)

    def reducer(self, word, counts):
        yield (word, sum(counts))


if __name__ == '__main__':
    MRWordFreqCount.run()
```

dog, [1, 1, 1, 1, 1, 1]

dog, 6

MINER & KASCH

Canonical Word Count

```
from mrjob.job import MRJob
import re

WORD_RE = re.compile(r"[\w']+")

class MRWordFreqCount(MRJob):

    def mapper(self, _, line):
        for word in WORD_RE.findall(line):
            yield (word.lower(), 1)

    def reducer(self, word, counts):
        yield (word, sum(counts))


if __name__ == '__main__':
    MRWordFreqCount.run()
```

cat, [1, 1, 1, 1, 1, 1, 1, 1]

cat, 8

MRJOB DEMO AND EXERCISE!

Other options

Hadoop Streaming – More manual but faster

Hadoopy, Dumbo, haven't seen commits in years, mrjob in the past 12 hours

Pydoop is main competitor (not in this list)

	Java	Streaming*	mrjob*	dumbo*	hadoopy*
FILE: bytes read	22,726,677,381	0.94	1.34	2.55	1.97
FILE: bytes written	33,468,535,411	0.93	1.35	2.57	1.99
HDFS: bytes read	21,934,848,598	1.00	1.00	1.00	1.00
HDFS: bytes written	7,629,045,090	1.00	0.99	1.00	1.06
Map output bytes	12,978,686,993	0.91	1.40	2.11	2.11
Reduce shuffle bytes	11,336,515,993	0.92	1.35	2.53	1.97
Reduce input records	428,755,439	1.04	1.00	1.46	1.04
Time spent all maps (ms)	14,256,288	1.37	5.98	2.39	3.76
Time spent in all reduces (ms)	4,348,716	1.76	8.91	6.14	4.86
CPU time (ms)	14,016,540	1.17	4.68	3.68	2.76
Job run time (s)	1,074	1.54	7.31	3.90	4.20

*Ratios are relative to Java values

http://blog.cloudera.com/blog/2013/01/a-guide-to-python-frameworks-for-hadoop/

Pydoop

- Write MapReduce jobs in Python!
- Uses Hadoop C++ Pipes, which should be faster than wrapping streaming
- Actively being worked on
- I'm not sure which is better

Pydoop Word Count

```
with open('stop.txt') as f:
    STOP_WORDS = set(l.strip() for l in f if not l.isspace())

def mapper(_, v, writer):
    for word in v.split():
        if word in STOP_WORDS:
            writer.count("STOP_WORDS", 1)
        else:
            writer.emit(word, 1)

def reducer(word, icounts, writer):
    writer.emit(word, sum(map(int, icounts)))
```

```
$ pydoop script wc.py hdfs_input hdfs_output --upload-
file-to-cache stop.txt
```

Pig

- Pig is a higher-level platform and language for analyzing data that happens to run MapReduce underneath

```
a = LOAD 'inputdata.txt';
b = FOREACH a GENERATE
        FLATTEN(TOKENIZE((chararray)$0)) as word;
c = GROUP b BY word;
d = FOREACH c GENERATE group, COUNT(c);
STORE d INTO 'wc';
```

Pig UDFs

Users can write *user-defined functions* to extend the functionality of Pig

Can use jython (faster) or cpython (access to more libs)

b = FOREACH a GENERATE revster(phonenum);

...

m = GROUP j BY username;

n = FOREACH m GENERATE group, sortedconcat(j.tags);

```
@outputSchema("tags:chararray")
def sortedconcat(bag):
    out = set()
    for tag in bag:
        out.add(tag)
    return '-'.join(sorted(out))
```

```
@outputSchema("rev:chararray")
def revstr(instr):
    return instr[::-1]
```

PIG DEMO AND EXERCISE!

snakebite

- A pure Python client
- Handles most NameNode ops (moving/renaming files, deleting files)
- Handles most DataNode reading ops (reading files, getmerge)
- Doesn’t handle writing to DataNodes yet
- Two ways to use: library and command line interface

snakebite - Library

```
from snakebite.client import Client

client = Client(”1.2.3.4", 54310, use_trash=False)

for x in client.ls(['/data']):
    print x

print ‘’.join(client.cat(‘/data/ref/refdata*.csv’))
```

Useful for doing HDFS file manipulation in data flows or job setups

Can be used to read reference data from MapReduce jobs

snakebite - CLI

```
$ snakebite get /path/in/hdfs/mydata.txt /local/path/data.txt

$ snakebite rm /path/in/hdfs/mydata.txt

$ for fp in `snakebite ls /data/new/`; do
    snakebite mv "/data/new/$fp" "/data/in/`date '+%Y/%m/%d/'$fp
  done
```

The “hadoop” CLI client is written in Java and spins up a new JVM every time (1-3 sec)

Snakebite doesn’t have that problem, making it good for lots of programmatic interactions with HDFS.

SNAKEBITE DEMO AND EXERCISE!

From the website:

Apache HBase is the Hadoop database, a distributed, scalable, big data store.

When Would I Use Apache HBase?

Use Apache HBase when you ***need random, realtime read/write access to your Big Data****. This project's goal is the hosting* ***of very large tables -- billions of rows X millions of columns*** *-- atop clusters of commodity hardware. Apache HBase is an open-source, distributed, versioned, non-relational database modeled after Google's Bigtable: A Distributed Storage System for Structured Data by Chang et al. Just as Bigtable leverages the distributed data storage provided by the Google File System, Apache HBase provides Bigtable-like capabilities on top of Hadoop and HDFS.*

Python clients

Starbase or **Happybase**
Uses the HBase Thrift gateway interface (slow)
Last commit 6 months ago
Appears to be fully featured

Not really there yet and have failed to gain community momentum. Java is still king.

From the website:

*Apache Spark is a **fast** and **general-purpose cluster computing system**. It provides high-level APIs in Scala, Java, and **Python** that make parallel jobs easy to write, and an optimized engine that supports general computation graphs. It also supports a rich set of higher-level tools including Shark (Hive on Spark), MLlib for machine learning, GraphX for graph processing, and Spark Streaming.*

In general, Spark is faster than MapReduce and easier to write than MapReduce

PySpark

- Spark's native language is Scala, but it also supports Java and Python
- Python API is always a tad behind Scala

- Programming in Spark (and PySpark) is in the form of chaining transformations and actions on RDDs
- RDDs are "Resilient Distributed Datasets"
- RDDs are kept in memory for the most part

PySpark Word Count Example

```
import sys
from operator import add
from pyspark import SparkContext

if __name__ == "__main__":
    if len(sys.argv) != 2:
        print >> sys.stderr, "Usage: wordcount <file>"
        exit(-1)
    sc = SparkContext(appName="PythonWordCount")
    lines = sc.textFile(sys.argv[1], 1)
    counts = lines.flatMap(lambda x: x.split(' ')) \
                  .map(lambda x: (x, 1)) \
                  .reduceByKey(add)
    output = counts.collect()
    for (word, count) in output:
        print "%s: %i" % (word, count)

    sc.stop()
```

PySpark Tutorial

PySpark tutorial tomorrow at 9am!

In this tutorial we will cover the basics of writing spark programs in python (initially from the pyspark shell, later with independent applications). We will also discuss some of the theory behind spark, and some performance considerations when using spark in a cluster.

https://us.pycon.org/2015/schedule/presentation/329/

HADOOP WITH PYTHON TUTORIAL

Donald Miner
@donaldpminer
dminer@minerkasch.com

PyCon 2015
April 9th 2015

MINER & KASCH